CORE
Metadata, citation and similar papers at core.ac.uk
Provided by NASA Technical Reports Server

N94-23697

AN OBJECT-ORIENTED APPROACH FOR PARALLEL SELF ADAPTIVE MESH REFINEMENT ON BLOCK STRUCTURED GRIDS[1]

S24-62
197156
P-15

Max Lemke[2] and Kristian Witsch
Mathematisches Institut der Universität Düsseldorf, Germany

Daniel Quinlan[2]
Computational Mathematics Group, University of Colorado, Denver

SUMMARY

Self-adaptive mesh refinement dynamically matches the computational demands of a solver for partial differential equations to the activity in the application's domain. In this paper we present two C++ class libraries, P++ and AMR++, which significantly simplify the development of sophisticated adaptive mesh refinement codes on (massively) parallel distributed memory architectures. The development is based on our previous research in this area. The C++ class libraries provide abstractions to separate the issues of developing parallel adaptive mesh refinement applications into those of parallelism, abstracted by P++, and adaptive mesh refinement, abstracted by AMR++. P++ is a parallel array class library to permit efficient development of architecture independent codes for structured grid applications, and AMR++ provides support for self-adaptive mesh refinement on block-structured grids of rectangular non overlapping blocks. Using these libraries the application programmers' work is greatly simplified to primarily specifying the serial single grid application, and obtaining the parallel and self-adaptive mesh refinement code with minimal effort.

Initial results for simple singular perturbation problems solved by self-adaptive multilevel techniques (FAC, AFAC), being implemented on the basis of prototypes of the P++/AMR++ environment, are presented. Singular perturbation problems frequently arise in large applications, e.g. in the area of computational fluid dynamics. They usually have solutions with layers which require adaptive mesh refinement and fast basic solvers in order to be resolved efficiently.

INTRODUCTION

The purpose of local mesh refinement during the solution of partial differential equations (PDEs) is to match computational demands to an application's activity: In a fluid flow problem this means that only regions of high local activity (shocks, boundary layers, etc.) can demand increased computational effort; regions of little flow activity (or interest) are more easily solved using only relatively little computational effort. In addition, the ability to adaptively tailor the computational mesh to the changing requirements of the application problem at runtime (e.g. moving fronts in time dependent problems) provides for much faster solution methods than static refinement or even uniform grid methods. Combined with increasingly powerful parallel computers that are becoming available, such methods allow for much larger and more comprehensive applications to be run. With local refinement methods, the greater disparity of scale introduced in larger applications can be addressed locally. Without local refinement, the resolution of smaller features in the applications domain can impose global limits either on the mesh size or the time step. The increased computational work associated with processing the global mesh cannot be readily offset even by the increased computational power of advanced parallel computers. Thus, local refinement is a natural part of the use of advanced massively parallel computers to process larger and more comprehensive applications.

---

[1]Revised and shortened version of [10]. This research has been supported by the National Aeronautics and Space Administration under grant number NASI-18606 and the German Federal Ministry of Research and Technology (BMFT) under PARANUSS, grant number ITR 900689.

[2]Part of this work belongs to the author's dissertation.

PRECEDING PAGE BLANK NOT FILMED
PAGE 344 INTENTIONALLY BLANK

Our experiments with different local refinement algorithms for the solution of the simple potential flow equation on parallel distributed memory architectures (e.g. [8]) demonstrates that, with the correct choice of solvers, performance of local refinement codes shows no significant sign of degradation as more processors are used. In contrast to conventional wisdom, the fundamental techniques used in our adaptive mesh refinement methods do not oppose the requirements for efficient vectorization and parallelization. However, the best choice of the numerical algorithm is highly dependent on its parallelization capabilities, the specific application problem and its adaptive grid structure, and, last but not least, the target architectures' performance parameters. Algorithms that are expensive on serial and vector architectures, but are highly parallelizable, can be superior on one or several classes of parallel architectures.

Our previous work with parallel local refinement, which was done in the C language to better allow access to dynamic memory management, has permitted only simplified application problems on non block structured composite grids of rectangular patches. The work was complicated by the numerical properties of local refinement, including self adaptivity and their parallelization capabilities like, for example, static and dynamic load balancing. In particular, the explicit introduction of parallelism in the application code is very cumbersome. Software tools for simplifying this are not available, e.g., existing grid oriented communication libraries (as used in [6]) are far too restrictive to be efficiently applied to this kind of dynamic problem. Thus, extending this code for the solution of more general complex fluid flow problems on complicated block structured grids is limited by the software engineering problem of managing the large complexities of the application problem, the numerical treatment of self-adaptive mesh refinement, complicated grid structures, and explicit parallelization. The development of codes that are portable across different target architectures and that are applicable to not just one problem and algorithm, but to a larger class, is impossible under these conditions.

Our solution to this software difficulty presents abstractions as a means of handling the combined complexities of adaptivity, mesh refinement, the application specific algorithm, and parallelism. These abstractions greatly simplify the development of algorithms and codes for complex applications. As an example, the abstraction of parallelism permits the development of application codes (necessarily based on parallel algorithms as opposed to serial algorithms, whose data and computation structures do not allow parallelization) in the simplified serial environment, and the same code to be executed in a massively parallel distributed memory environment.

This paper introduces an innovative set of software tools to simplify the development of parallel adaptive mesh refinement codes for difficult algorithms. The tools are present in two parts, which form C++ class libraries and allow for the management of the great complexities described above. The first class library, P++ (short summary in Section 2, details in [10]), forms a data parallel superset of the C++ language with the commercial C++ array class library M++ (Dyad Software Corporation). A standard C++ compiler is used with no modifications of the compiler required. The second set of class libraries, AMR++ (Section 3), forms a superset of the C++/M++, or P++, environment and further specifies the combined environment for local refinement (or parallel local refinement). In Section 4 we introduce multilevel algorithms that allow for the introduction of self-adaptive mesh refinement (Asynchronous) Fast Adaptive Composite Methods (FAC and AFAC)). In Section 5, we present first results for a simple singular perturbation problem that has been solved using FAC and AFAC algorithms being implemented on the bases of AMR++ and P++ prototypes. This problem serves as a good model problem for complex fluid flow applications, because several of the properties that are related to self-adaptive mesh refinement are already present in it.

We are particularly grateful to Steve McCormick, without whose support this joint work would not have been possible, and to the people at the Federal German Research Center Jülich (KFA) for their generous support in letting us use their iPSC/860 environment. In addition we would like to thank everybody who discussed P++ or AMR++ with us or in any other way supported our work.

P++, A PARALLEL ARRAY CLASS LIBRARY FOR STRUCTURED GRIDS

P++ is an innovative, robust, and architecture-independent array class library that simplifies the development of efficient parallel programs for large scale scientific applications by abstracting parallelism. The target machines are current and evolving massively parallel distributed memory multiprocessor systems (e.g. Intel iPSC/860 and PARAGON, Connection Machine 5, Cray MPP, IBM RS 6000 networks) with different types of node architectures (scalar, vector, or superscalar). Through the use of portable communication and tool libraries (e.g. EXPRESS, ParaSoft Corporation), the requirements of shared memory computers are also addressed. The P++ parallel array class library is implemented in standard C++ using the serial M++ array class library, with absolutely no modification of the compiler. P++ allows for software development in the preferred serial environment, and such software to be efficiently run, unchanged, in all target environments. The runtime support for parallelism is both completely hidden and dynamic so that array partitions need not be fixed during execution. The added degree of freedom presented by parallel processing is exploited by use of an optimization module within the array class interface. For detail, please refer to [10].

Application class: The P++ application class is currently restricted to structured grid-oriented problems, which form a primary problem class currently represented in scientific supercomputing. This class is represented by dimensionally independent block structured grids (1D - 4D) with rectangular or logically rectangular grid blocks. The M++ array interface, which is also used as the P++ interface and whose functionality is similar to the array features of Fortran 90, is particularly well suited to express operations on grid blocks to the compiler and to the P++ environment at runtime.

Programming Model and Parallelism: P++ is based on a Single Program Multiple Data Stream (SPMD) programming model, which consists of executing one single program source on all nodes of the parallel system. Its combination with the Virtual Shared Grids (VSG) model of data parallelism (a restriction of virtual shared memory to structured grids, whose communication is controlled at runtime) is essential for the simplified representation of the parallel program using the serial program and hiding communication within the grid block classes. Besides different grid partitioning strategies, two communication update principles are provided and automatically selected at runtime: Overlap Update for very efficient nearest neighbor grid element access of aligned data and VSG Update for general grid (array) computations. By use of local partitioning tables, communication patterns are derived at runtime, and the appropriate send and receive messages of grid portions are automatically generated by P++ selecting the most efficient communication models for each operation. As opposed to general Virtual Shared Memory implementations, VSG allows for obtaining similar parallel performance as for codes based on the traditionally used explicit Message Passing programming model. Control flow oriented functional parallelism until now is not particularly supported in P++. However, a cooperation with the developers of CC++ ([4]) is planned.

Summary of P++ Features:

- Object oriented indexing of the array objects simplifies development of serial codes by removing error prone explicit indexing common to *for* or *do* loops.
- Algorithm and code development takes place in a serial environment. Serial codes are re-compilable to run in parallel without modification.
- P++ codes are portable between different architectures. Vectorization, parallelization and data partitioning are hidden from the user, except for optimization switches.
- P++ application codes exhibit communication as efficiently as codes with explicit message passing. With improved C++ compilers and an optimized implementation of M++, single node performance of C++ with array classes has the potential to approximate that of Fortran.

Current State, Performance Issues and Related Work: The P++ prototype is currently implemented on the bases of the AT&T C++ C-Front precompiler using the Intel NX-2 communication library (or, on an experimental basis, an EXPRESS-like portable communication library from Caltech). Current versions are running on the Intel iPSC/860 Hypercube, the Intel Simulator, SUN workstations, the Cray 2, and IBM PCs. The prototype contains all major concepts described above. At several points, without loss of generality, its functionality is restricted to the needs within our own set of test problems (3D multigrid codes and FAC/AFAC codes).

The feasibility of the approach has been proven by the successful implementation and use of our set of test problems on the basis of P++, in particular, the very complex AMR++ class library. The results that have been obtained with respect to parallel efficiency, whose optimization was one of the major goals of the P++ development, are also very satisfying: Comparisons for P++ and Fortran with message passing based test codes, respectively, have shown that the number of messages and the amount of communicated data is roughly the same. Thus, besides a negligible overhead, similar parallel efficiency can be achieved. With respect to single node performance, only little optimization has been done. The major reason is that the used system software components (AT&T C++ C-Front precompiler 2.1, M++) are not very well optimized for the target machines. However, our experiences with C++ array language class libraries on workstations and on the Cray Y-MP (in collaboration with Sandia National Laboratories: about 90% of the Fortran vector performance is achieved) are very promising: With new optimized system software versions, Fortran performance can be approximated. Therefore, altogether, we expect the parallel performance for P++ based codes to be similar to that obtained for optimized Fortran codes with explicit message passing.

AMR++, AN ADAPTIVE MESH REFINEMENT CLASS LIBRARY

AMR++ is a C++ class library that simplifies the details of building self-adaptive mesh refinement applications. The use of this class library significantly simplifies the construction of local refinement codes for both serial and parallel architectures. AMR++ has been developed in a serial environment using C++ and the M++ array class interface. It runs in a parallel environment, because M++ and P++ share the same array interface. The nested set of abstractions provided by AMR++ uses P++ at its lowest level to provide architecture independent support. Therefore, AMR++ inherits the machine targets of P++, and, thus, has a broad base of machines on which to run. The efficiency and performance of AMR++ is mostly dependent on the efficiency of M++ and P++, in the serial and parallel environments respectively. In this way, the P++ and AMR++ class libraries separate the abstractions of local refinement and parallelism to significantly ease the development of parallel adaptive mesh refinement applications in an architecture independent manner. The AMR++ class library represents work which combines complex numerical, computer science, and engineering application requirements. Therefore, the work naturally involves compromises in its initial development. In the following sections, the features and current restrictions of the AMR++ class library are summarized.

Block Structured Grids Features and Restrictions: The target grid types of AMR++ are 2D and 3D block structured with rectangular or logically rectangular blocks. On the one hand, they allow for a very good representation of complex internal geometries introduced through local refinement in regions with increased local activity. This flexibility of local refinement block structured grids equally applies to global block structured grids that allow for matching complex external geometries. On the other hand, the restriction to structures of rectangular blocks, as opposed to fully unstructured grids, allows for the application of the VSG programming model of P++ and, therefore, is the foundation for good efficiency and performance in distributed environments, which is one of the major goals of the P++/AMR++ development. Thus, we believe that block structured grids are the best compromise between full generality of the grid structure and efficiency in a distributed parallel environment. The application class forms a broad cross section of important scientific applications.

In the following, the global grid is the finest uniformly discretized grid that covers the whole physical domain. Local refinement grids are formed from the global grid, or recursively from refinement grids, by standard refinement with $h_{fine} = \frac{1}{2} h_{coarse}$ in each coordinate direction. Thus, boundary lines of block structured refinement grids always match grid lines on the underlying discretization level. The construction of block structured grids in AMR++ has some practical limitations that simplify the design and use of the class libraries. Specifically, grid blocks at the same level of discretization cannot overlap. Block structures are formed by distinct or connected rectangular blocks that share their boundary points (block interfaces) at those places where they adjoin each other. Thus, a connected region of blocks forms a block structured refinement grid. It is possible that one refinement level consists of more than one disjunct block structured refinement grid. In the dynamic adaptive refinement procedure, refinement grids can be automatically merged, if they adjoin each other.

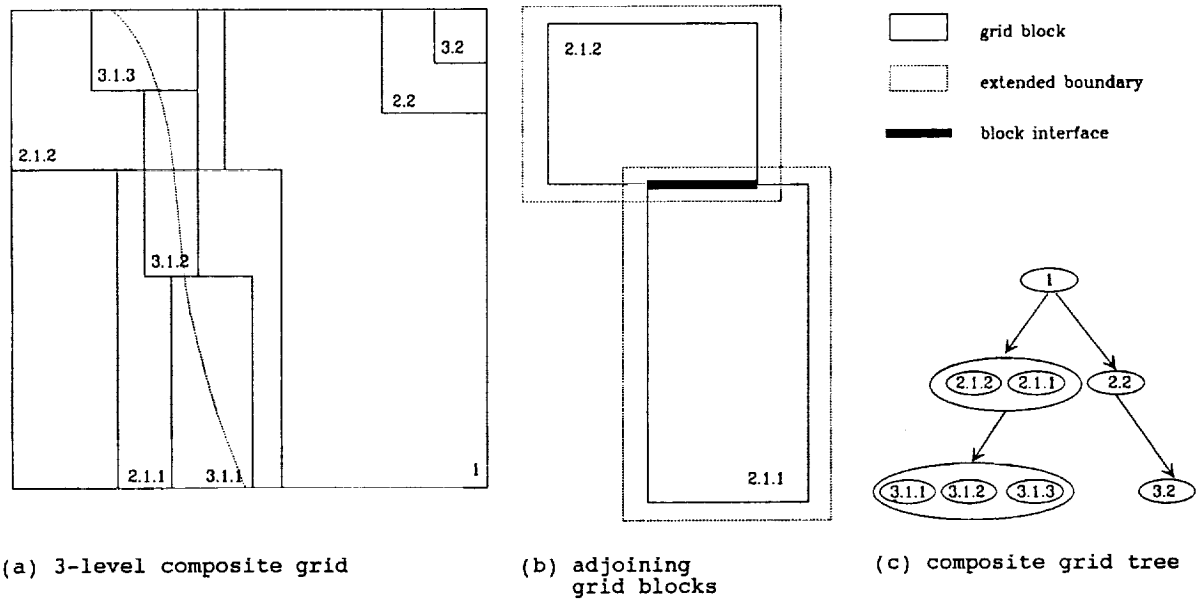

Figure 1: Example of a composite grid, its composite grid tree, and a cut out of 2 blocks with their extended boundaries and interface.

In Figure 1 (a), an example for a composite grid is illustrated: The composite grid shows a rectangular domain within which we center a curved front and a corner singularity. The grid blocks are ordered lexicographically: the first digit represents the level, the second digit the connected block structured refinement grid, and the third digit the grid block. Such problems could represent the structure of shock fronts or multi-fluid interfaces in fluid flow applications: In oil reservoir simulations, for example, the front could be an oil water front moving with time and the corner singularity could be a production well. In this specific example, the front is refined with two block structured refinement grids: the first grid on refinement level 2 is represented by grid blocks 2.1.1 and 2.1.2, and the second grid on level 2 by grid blocks 3.1.1, 3.1.2 and 3.1.3. In the corner on each of the levels, a single refinement block is introduced.

For ease of implementation, in the AMR++ prototype the global grid must be uniform. This simplification of the global geometry was necessary in order to be able to concentrate on the major issues of this work, namely, the implementation of local refinement and self adaptivity in an object-oriented environment. This restriction is not critical and can be eased in future versions of the prototype. Aside from implementation issues, some additional functionality must be made available:

- For implicit solvers, the resulting domain decomposition of the global grid may require special capabilities within the single grid solvers (e.g., multigrid solvers for block structured grids with adequate smoothers, such as inter-block line or plane relaxation methods).

- The block structures in the current AMR++ prototype are defined only by the needs of local refinement of a uniform global grid. This restriction allows them to be Cartesian. More complicated structures as they result from difficult non Cartesian external geometries (e.g., holes; see [11]) currently are not taken into consideration. An extension of AMR++, however, is principally possible. The wide experience for general 2D block structured grids that has been gained at GMD [11] can form a basis for these extensions. Whereas our work is comparably simple in 2D, because no explicit communication is required, extending the GMD work to 3D problems is very complex.

Some Implementation Issues: In the following, some implementation issues are detailed. They also demonstrate the complexity of a proper and efficient treatment of block structured grids and adaptive refinement. AMR++ takes care of all of these issues, which would otherwise have to be handled explicitly at the application level.

- Dimensional independence and multi-indexing: The implementation of most features of AMR++ and its user interface is dimensionally independent. Being derived from user requirements, on the lowest level, the AMR++ prototype is restricted to 2D and 3D applications. This, however, is a restriction that can easily be removed.

  One important means by which dimensional independence is reached, is multi-dimensional indices (multi-indices), which contain one index for each coordinate direction. On top of these multi-indices are index variants defined for each type of sub-block (interior, interior and boundary, boundary only, ...), which contain multiple multi-indices. For example, for addressing the boundary of a 3D block (non-convex), one multi-index is needed for each of the six planes. In order to avoid special treatment of physical boundaries, all index variants are defined twice, including and excluding the physical boundary, respectively. All index variants, several of them also including extended boundaries (see below), are precomputed at the time when a grid block is allocated. In the AMR++ user interface and in the top level classes, only index variants or indicators are used and, therefore, allow a dimensionally independent formulation, except for very low level implementations.

- Implementation of block structured grids: The AMR++ grid block objects consist of the interior, the boundary, an extended boundary of a grid block, and links that are formed between adjacent pairs of grid block objects. The links contain P++ array objects that do not consist of actual data, but serve as views (subarrays) of the overlapping parts of the extended boundary between adjacent grid block objects. The actual boundaries that are shared between different blocks (block interfaces) are very complex structures that are represented properly in the grid block objects. For example, in 3D, interfaces between blocks are 2D planes, those between plane-interfaces are 1D-line interfaces, and, further, those between line-interfaces are points (zero-dimensional).

  In Figure 1 (b), grid blocks 2.1.1 and 2.1.2 of the composite grid in Figure 1 (a) are depicted including their block interface and their extended boundary. The regular lines denote the outermost line of grid points of each block. Thus, with an extended boundary of two, there is one line of points between the block boundary line and the dashed line for the extended boundary. In its extended boundary, each grid block has views of the values of the original grid points of its adjoining neighboring block. This way it is possible to evaluate stencils on the interface and, with an extended boundary width of two, to also define a coarse level of the block structured refinement grid in multigrid sense.

- Data structures and iterators: In AMR++, the composite grid is stored as a tree of all refinement grids, with the global grid being the root. Block structured grids are stored as lists of blocks (for ease of implementation; collections of blocks would be sufficient in most cases).

In Figure 1 (c), the composite grid tree for the example composite grid in Figure 1 (a) is illustrated.

The user interface for doing operations on these data structures are so-called iterators. For example, for an operation on the composite grid (e.g., zeroing each level or interpolating a grid function to a finer level), an iterator is called that traverses the tree in the correct order. (preorder, postorder, no order). This iterator as arguments takes the function to be executed and two indicators that specify the physical boundary treatment and the type of sub grid to be treated. The iteration starts at the root and recursively traverses the tree. For doing an operation (e.g. Jacobi relaxation) on a block structured grid, iterators are available, that process the list of blocks and all block interface lists. They take arguments similar to those for the composite grid tree iterators.

Object-Oriented Design and User Interface: The AMR++ class libraries are customizable by using the object oriented features of C++. For example, in order to obtain efficiency in the parallel environment, it may be necessary to introduce alternate iterators that traverse the composite grid tree or the blocks of a refinement region in a special order. This is implemented by alternate use of different base classes in the serial and parallel environment. The same is true for alternate composite grid cycling strategies as, for example, needed in AFAC, in contrast to FAC algorithms (Section 4). Application specific parts of AMR++, such as the single grid solvers or criteria for adaptivity, which have to be supplied by the user, are also simply specified through substitution of alternate base classes: A pre-existing application (e.g., problem setup and uniform grid solver) uses AMR++ to extend its functionality and to build an adaptive mesh refinement application. Thus, the user supplies a solver class and some additional required functionality (refinement criteria, ...) and uses the functionality of the highest level AMR++ ((Self_)Adaptive_)Composite_Grid class to formulate his special algorithm or to use one of the supplied PDE solvers. In the current prototype of AMR++, FAC and AFAC based solvers (Section 4) are supplied. If the single grid application is written using P++, then the resulting adaptive mesh refinement application is architecture independent, and so can be run efficiently in a parallel environment.

The design and interface of AMR++ is object-oriented and the implementation of our prototype extensively uses features like encapsulation and inheritance: The abstraction of self-adaptive local refinement, which involves the handling of many issues (including memory management, interface for application specific control, dynamic adaptivity, and efficiency), is reached through grouping these different functionalities in several interconnected classes. For example, memory management is greatly simplified by the object oriented organization of the AMR++ library: Issues such as lifetime of variables are handled automatically by the scoping rules for C++, so memory management is automatic and predictable. Also, the control over construction of the composite grid is intuitive and natural: The creation of composite grid objects is similar to the declaration of floating point or integer variables in procedural languages like Fortran and C. The user basically formulates a solver by allocating one of the predefined composite grid solver objects, or by formulating it on the basis of the composite grid objects and associated iterators and by supplying the single grid solver class.

Although not part of the current implementation of AMR++, C++ introduces a template mechanism in the latest standardization of the language, which is only just beginning to be part of commercial products. The general purpose of this template language feature is to permit class libraries to access user specified base types. For AMR++, for example, the template feature could be used to allow the specification of the base solver and adaptive criteria for the parallel adaptive local refinement implementation. In this way, the construction of an adaptive local refinement code from the single grid application on the basis of the AMR++ class library can become even simpler and cleaner. The object-oriented design of interconnected classes will not be further discussed. The reader is referred instead to [10] and [7].

Static and Dynamic Adaptivity, Grid Generation: In the current AMR++ prototype, static adaptivity is fully implemented. The user can specify a composite grid either interactively or by

some input file: For each grid block, AMR++ needs its global coordinates and the parent grid block. Block structured local refinement regions are formed automatically by investigating neighboring relationships. In addition, the functionalities for adding and deleting grid blocks under user control are available within the Adaptive_Composite_Grid object of AMR++.

Recently, dynamic adaptivity has been a subject of intensive research. Initial results are very promising, and some basic functionality has been included in the AMR++ prototype: Given a global grid, a flagging criteria function, and some stopping criteria, the Self_Adaptive_Composite_Grid object contains the functionality for iteratively solving on the actual composite grid and generating a new discretization level on top of the respective finest level. Building a new composite grid level works as follows:

1. The flagging criteria delivers an unstructured collection of flagged points in each grid block. For representing grid block boundaries, all neighboring points of flagged points are also flagged.

2. The new set of grid blocks to contribute to the refinement level (gridding) is built by applying a smart recursive bisection algorithm similar to the one developed in [2]: If building a rectangle around all flagged points of the given grid block is too inefficient, it is bisected in the longer coordinate direction and new enclosing rectangles are computed. The efficiency of the respective fraction is measured by the ratio of flagged points to all points of the new grid block. In the following tests, 75% is used. This procedure is repeated recursively if any of the new rectangles is also inefficient. Having the goal of building the rectangles as large as possible within the given efficiency constraint, the choice of the bisection point (splitting in halves is too inefficient because it results in very many small rectangles) is done by a combination of signatures and edge detection. A detailed description of this method reaches beyond the scope of this paper, so the reader is referred to [2] or [7].

3. Finally, the new grid blocks are added to the composite grid to form the new refinement level. Grouping these blocks into connected block structured grids is done the same way as it is done in the static case.

This flagging and gridding algorithm has the potential for further optimization: The bisection method can be further improved, and a clustering and merging algorithm could be applied. This is especially true for refinement blocks of different parent blocks that could form one single block with more than one parent. Internal to AMR++, this kind of parent / child relationship is supported. The results in Section 5, however, show that the gridding already is quite good. The number of blocks that are constructed automatically is only slightly larger ($< 10\%$) than a manual construction would deliver. A next step in self-adaptive refinement would be to support time dependent problems whose composite grid structure changes dynamically with time (e.g., moving fronts). In this case, in addition to adding and deleting blocks, enlarging and diminishing blocks must be supported. Though some basic functionality and the implementation of the general concept is already available, this problem has not yet been further pursued.

Current State and Related Work: The AMR++ prototype is implemented using M++ and the AT&T Standard components class library to provide standardized classes (e.g., linked list classes). Through the shared interface of M++ and P++, AMR++ inherits all target architectures of P++. The prototype has been successfully tested on SUN workstations and on the Intel iPSC/860, where it has proved its full functionality with respect to parallelization. Taking into account the large application class of AMR++, there are still several insufficiencies and restrictions, as well as a large potential for optimization. For parallel environments, e. g., efficiently implementing self-adaptivity, including load (re)balancing, requires further research. In addition, the iterators that are currently available in AMR++, though working in a parallel environment, are best suited for serial environments. Special parallel iterators that, for example, support functional parallelism on the internal AMR++ level would have to be provided. Until now, AMR++ has been successfully used as a research tool for the algorithms and model problems described in the next two sections.

However, AMR++ provides the functionality to implement much more complicated application problems.

Concerning parallelization, running AMR++ under P++ on the Intel iPSC/860 has proven its full functionality. Intensive optimization, however, has only been done within P++. AMR++ itself offers a large potential for optimization.

To the authors' knowledge, the AMR++ approach is unique. There are several other developments in this area (e.g. [11]), but they either address a more restricted class of problems or are restricted to serial environments.

MULTILEVEL ALGORITHMS WITH ADAPTIVE MESH REFINEMENT

The fast adaptive composite grid method (FAC, [12]), which was originally developed from and is very similar to the Multi-Level Adaptive Technique (MLAT, [3]), is an algorithm that uses uniform grids, both global and local, to solve partial differential equations. This method is known to be highly efficient on scalar or single processor vector computers, due to its effective use of uniform grids and multiple levels of resolution of the solution. On distributed memory multiprocessors, methods like MLAT or FAC benefit from their tendency to create multiple isolated refinement regions, which may be effectively treated in parallel. However, for several problem classes, they suffer from the way in which the levels of refinement are treated sequentially in each region. Specifically, the finer levels must wait to be processed until the coarse-level approximations have been computed and passed to them; conversely, the coarser levels must wait until the finer level approximations have been computed and used to correct their equations. Thus, the parallelization potential of these "hierarchical" methods is restricted to intra-level parallelization.

The asynchronous fast adaptive composite method (AFAC) eliminates this bottleneck of parallelism. Through a simple mechanism used to reduce inter-level dependencies, individual refinement levels can be processed by AFAC in parallel. The result is that the convergence rate for AFAC is the square root of that for FAC. Therefore, since both AFAC and FAC have roughly the same number of floating point operations, AFAC requires twice the serial computational time as FAC, but AFAC allows for the introduction of inter-level parallelization.

As opposed to the original development of FAC and AFAC, in this paper, the modified algorithms known as FACx and AFACx are discussed and used. They differ in the treatment of the refinement levels. Whereas in FAC and AFAC, a rather accurate solution is computed (e.g., one MG V-cycle), FACx uses only a couple of relaxations. AFACx uses a two-grid procedure (of FMG-type) on the refinement level and its standard coarsening with several relaxations on each of these levels. Experiments and some theoretical observations show that all of the results that have been obtained for FAC and AFAC also hold for FACx and AFACx (see [14]). In the following, FAC and AFAC always denote the modified versions (FACx and AFACx).

Numerical algorithms: Both FAC (MLAT) and AFAC consist of two basic steps, which are described loosely as follows:

1. Smoothing phase: Given the solution approximation and composite grid residuals on each level, use relaxation or some restricted multigrid procedure to compute a correction local to that level (a better approximation is required on the global grid, the finest uniform discretization level).

2. Level transition phase: Combine the local corrections with the global solution approximation, compute the global composite grid residual, and transfer the local components of the approximation and residual to each level.

The difference between MLAT and FAC on the one hand and AFAC on the other hand is in the order in which the levels are processed and in the details of how they are combined:

- FAC and MLAT can roughly be viewed as standard multigrid methods with mesh refinement and a special treatment of the interfaces between the refinement levels and the underlying coarse level. In FAC and MLAT the treatment of the refinement levels is hierarchical. Theory on FAC is based on its interpretation as a multiplicative Schwarz Alternating Method or as a block relaxation method of Gauss-Seidel type.

  FAC and MLAT mainly differ by their motivation. Whereas it is the goal of FAC to compute a solution for the composite grid (grid points of the composite grid are all the interior points of the respective finest discretization level), the major goal of MLAT is to get the best possible solution on a given uniform grid (with using local refinement). Thus, in FAC, coarse levels of the composite grid serve for the computation of corrections. Therefore, FAC was originally formulated as a correction scheme (CS). The MLAT formulation requires a full approximation scheme (FAS), because coarse levels serve as correction levels for the points covered by finer levels. MLAT was first developed using finite difference discretization, whereas for FAC finite volume discretizations were used. However, they are closely related and in many problems lead to the same stencil representation. This is true except perhaps for the interface points, where finite volume discretizations generally lead to conservative discretizations (FAC), whereas finite difference discretizations do not (MLAT). Instead, in MLAT, usually a higher order interpolation is used on the interface. Other than this exception, because of the modification of the original FAC algorithm as discussed above, there is no difference in the treatment of the refinement levels between the original MLAT algorithm and the modified FAC algorithm that is discussed in this paper. It can be shown ([7]) that an FAS version of FAC with a special choice of the operators on the interface is equivalent to the originally developed Multilevel Adaptive Technique (MLAT).

- AFAC on the other hand consists of the same discretization and operators as FAC, but a decoupled and asynchronous treatment of the refinement levels in the solution phase, which dominates the arithmetic work in the algorithm. Theory on AFAC can be based on its interpretation as an additive Schwarz Alternating Method or as a block relaxation method of Jacobi type.

Theory in [12] shows that, under appropriate conditions, the convergence factors of FAC and AFAC have the relation $\rho_{AFAC} = \sqrt{\rho_{FAC}}$. This implies that two cycles of AFAC are roughly equivalent to one cycle of FAC. If the algorithmic components are chosen slightly different than for the convergence analysis or if applied to singular perturbation problems as discussed in the next section, experiences show that AFAC is usually better than as suggested by the above formula: In several cases, the convergence factor of AFAC shows only a slight degradation of the FAC rate (Section 5).

Parallelization — an Example for the Use of P++/AMR++: By example, we demonstrate some of the features of AMR++ and examples for the support of P++ for the design of parallel block structured local refinement applications on the basis of FAC and AFAC algorithms.

In a parallel environment, partitioning the composite grid levels becomes a central issue in the performance of composite grid solvers. In Figure 2, two different partitioning strategies that are supported within P++/AMR++ are illustrated for the composite grid in Figure 2. For ease of illustration, grid blocks 2.2 and 2.3 are not included. The so-called FAC partitioning in Figure 2 (b) is typical for implicit and explicit algorithms, where the local refinement levels have to be treated in a hierarchical manner (FAC, MLAT,...). The so-called AFAC partitioning in Figure 2 (a) can be optimal for implicit algorithms that allow an independent and asynchronous treatment of the refinement levels. In the case of AFAC, however, it must be taken into consideration that this partitioning is only optimal for the solution phase, which dominates the arithmetic work of the algorithm. The efficiency of the level transition phase, which is based on the same hierarchical structure as FAC and which can eventually dominate the aggregate communication work of the algorithm, highly depends on the architecture and the application (communication / computation ratio, single node (vector) performance, message latency, transfer rate, congestion, ...). For

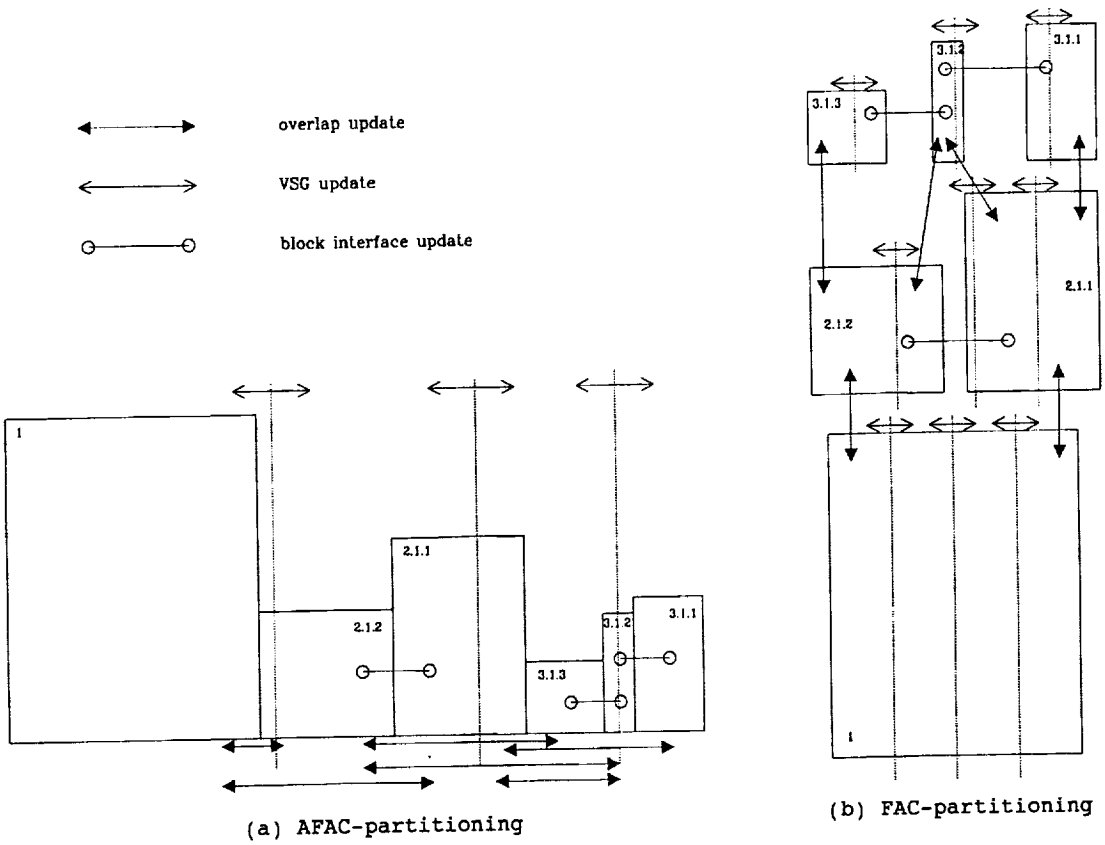

Figure 2: Parallel multilevel local refinement algorithms on block structured grids — an example for the use of AMR++ and the hidden interaction of the P++ communication models.

determining whether AFAC is better than FAC in a parallel environment, the aggregate efficiency and performance of both phases and the relation of the convergence rates must be properly evaluated. For more detail, see [10] and [7]. Both types of partitioning are supported in the P++/AMR++ environment.

Solvers used on the individually partitioned composite grid levels make use of overlap updates within P++ array expressions, which automatically provide communication as needed. The inter-grid transfers between local refinement levels, typically located on different processors, rely on VSG updates. The VSG updates are also provided automatically by the P++ environment. Thus, the underlying support of parallelism is isolated in P++ through either overlap update or VSG update, or a combination of both, and the details of parallelism are isolated away from the AMR++ application. The block structured interface update is handled in AMR++. However, communication is hidden in P++ (mostly the VSG update).

RESULTS FOR SINGULAR PERTURBATION PROBLEMS

Use of the tools described above is now demonstrated with initial examples. The adaptivity provided by AMR++ is necessary in case of large gradients or singularities in the solution of the PDE. They may be due to rapid changes in the right-hand side or coefficients of the PDE, corners in the domain, or singular perturbations. Here, the first and the last case will be examined on the basis of model problems.

Singularly perturbed PDEs represent the modelling of physical processes with relatively small diffusion (viscosity) and dominating convection. They may occur as a single equation or within

systems of complex equations, e.g., as the momentum equations within the Navier-Stokes or, in addition, as supplementary transport equations in the Boussinesq system of equations. Here, we merely treat a single equation. However, we only use methods that generalize directly to more complex situations. Therefore, we do not rely on the direct solution methods provided by downstream or ILU relaxations for simple problems with pure upstream discretization. The latter are not direct solution methods for systems of equations. Further, these types of flow direction dependent relaxations are not efficiently parallelizable in the case of only a few relaxations as is usually used in multilevel methods. This in particular holds on massively parallel systems.

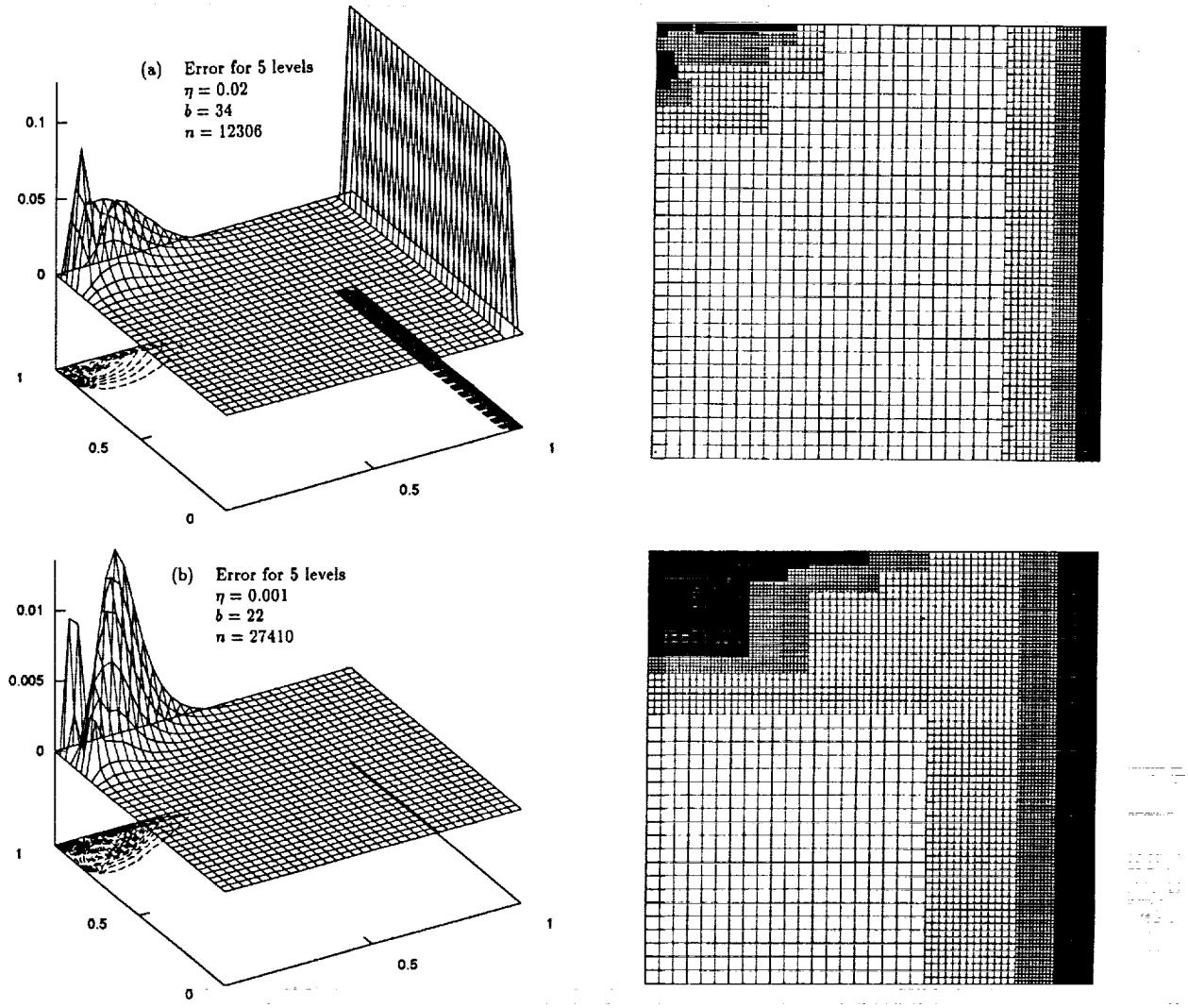

Figure 3: Results for a singular perturbation problem: Plots of the error and composite grid, with two different choices of the accuracy η in the self-adaptive refinement process.

Model Problem and Solvers: Numerical results have been obtained for the model problem

$$-\varepsilon\Delta u + au_x + bu_y = f \qquad \text{on } \Omega = (0,1)^2$$

with Dirichlet boundary conditions on $\partial\Omega$ and $\varepsilon = 0.00001$. This problem serves as a good model for complex fluid flow applications, because several of the properties that are related to self-adaptive mesh refinement are already present in this simple problem. The equation is discretized using

isotropic artificial viscosity (diffusion):

$$\begin{aligned} L_h &:= -\varepsilon_h \Delta_h + aD_{2h,x}u + bD_{2h,y}u \text{ with } \Delta_h = D_{h,x}^2 + D_{h,y}^2 \\ \varepsilon_h &:= \max\{\varepsilon, \beta h \max\{|a|,|b|\}/2\} \end{aligned}$$

The discrete system is solved by multilevel methods – MG on the finest global grid and FAC or AFAC on composite grids with refinement. For the multigrid method, it is known that, with artificial viscosity, the two-grid convergence factor (spectral radius of the corresponding iteration matrix) is bounded below by 0.5 (for $h \to 0$). Therefore, multilevel convergence factors converge to 1.0 with an increasing number of levels. In [5], a multigrid variant which shows surprisingly good convergence behavior has been developed: MG convergence factors stay far below 0.5 (with three relaxations on each level). Here, essentially this method is used, which is described as follows:

- Discretization with additional isotropic artificial viscosity using $\beta = 3$ on the finest grid m and $\beta_{l-1} = 1/2\ (\beta_l + 1/\beta_l)$ for coarser grids $l = m-1, m-2, \ldots,$
- MG components: odd/even relaxation, non-symmetric transfer operators corresponding to linear finite elements. These components fulfil the Galerkin condition for the Laplacian.

Anisotropic artificial viscosity may also be used, but generally requires (parallel) zebra line relaxation, which has not yet been fully implemented.

For FAC and AFAC, the above MG method with V(2,1) cycling is used as a global grid solver. On the refinement levels, three relaxations are performed, and $\beta = 3$ is chosen on refinement grids.

Convergence Results: In Table 1, several convergence factors for FAC, AFAC, and, for comparison, for MG are shown. The finest grids have mesh sizes of $h = 1/64$ or $h = 1/512$, respectively. For FAC and AFAC, the global grid has the mesh size $h = 1/32$, the (predetermined) fine block always covers 1/2 of the parent coarse block along the boundary layer. The following conclusions can be drawn:

- For MG, the results are as expected. In the case of FAC and AFAC, the choice of β has to be further investigated.
- V cycles are used; W or F cycles would yield better convergence rates but worse parallel efficiency.
- If $\rho(FAC)$ is small, the expected result $\rho(AFAC) \approx \sqrt{\rho(FAC)}$ can be observed, otherwise $\rho(FAC) \approx \rho(AFAC) \ll \sqrt{\rho(FAC)}$.

	Poisson		SPP: $\beta = 3$		SPP: $\beta = 1$	
h	1/64	1/512	1/64	1/512	1/64	1/512
MG-V	0.14	0.14	0.17	0.30	0.18	0.50
FAC	0.17	0.18	0.30	0.65	0.30	0.80
AFAC	0.40	0.41	0.41	0.67	0.45	0.95

Table 1: Convergence factors for a singular perturbation problem (SPP: $a = b = 1, \varepsilon = 0.00001$) and, for comparison, for Poisson's equation.

Self-Adaptive Mesh Refinement Results: More interesting for the goal of this paper are applications of the self-adaptive process. As opposed to the convergence rates, they do not depend

only on the PDE, but also on the particular solution. The results in this paper have been obtained for the exact solution

$$u(x) = \frac{e^{(x-1)/\varepsilon} - e^{-1/\varepsilon}}{1 - e^{-1/\varepsilon}} + \frac{1}{2}\, e^{-100(x^2+(y-1)^2)},$$

which has a boundary layer for $x = 1, 0 \le y \le 1$ and a steep hill around $x = 0, y = 1$. In order to measure the error of the approximate solution, a discrete approximation to the L_1 error norm is used. This is appropriate for this kind of problem: For solutions with discontinuities of the above type, one can observe 1st order convergence only with respect to this norm (no convergence in the L_∞ norm, order 0.5 in the L_2 norm).

The results have been obtained using the flagging criteria

$$h^f \, [\beta h \max\{|a|, |b|\} \, (|D^2_{h,x} u| + |D^2_{h,y} u|)] \;\ge\; \eta$$

with a given value of η. For $\varepsilon < \varepsilon_h$, the second factor is an approximation to the lowest order error term of the discretization. Based on experiments, $f = 1$ is a good choice. Starting with the global grid, the composite grid is self-adaptively built on the basis of the flagging and gridding algorithm described in Section 3.

	MG-V uniform		FAC $\eta = 0.02$			FAC $\eta = 0.01$			FAC $\eta = 0.001$		
h	e	n	e	n	b	e	n	b	e	n	b
1/32	0.0293	961	0.0293	961	1	0.0293	961	1	0.0293	961	1
1/64	0.0159	3969	0.0160	1806	4	0.0160	1967	4	0.0159	2757	3
1/128	0.0083	16129	0.0089	3430	10	0.0087	3971	10	0.0083	6212	7
1/256	0.0043	65025	0.0056	6378	19	0.0051	7943	16	0.0043	13473	12
1/512	0.0023	261121	0.0073	12306	34	0.0044	15909	30	0.0023	27410	22

Table 2: Accuracy (L1-norm e) vs. the number of grid points (n) and the number of blocks (b) for MG-V on a uniform grid and FAC on self-adaptively refined composite grids.

In Table 2, the results for MG and FAC are presented for three values of η. In Figure 3, two of the corresponding block structured grids are displayed. The corresponding error plots give an impression of the error distribution restricted from the composite grid to the global uniform grid. Thus, larger errors near the boundary layer are not visible. The results allow the following conclusions:

- In spite of the well known difficulties in error control of convection dominated problems, the grids that are constructed self-adaptively are reasonably well suited to the numerical problem.
- As long as the accuracy of the finest level is not reached, the error norm is approximatively proportional to η. As usual in error control by residuals, with the norm of the inverse operator being unknown, the constant factor is not known.
- If the refinement grid does not properly match the local activity, convergence rates significantly degrade and the error norm may even increase.
- Additional tests have shown that, if the boundary layer is fully resolved with an increased number of refinement levels, the discretization order, as expected, changes from one to two.
- The gridding algorithm is able to treat very complicated refinement structures efficiently: The number of blocks that are created is nearly minimal (compared to hand coding).

- Though this example needs relatively large refinement regions, the overall gain by using adaptive grids is more than 3.5 (taking into account the different number of points and the different convergence rates). For pure boundary layer problems, factors larger than 10 have been observed.
- These results have been obtained in a serial environment. AMR++, however, has been successfully tested in parallel. For performance and efficiency considerations, see Sect. 2 and 3.

References

[1] Balsara, D.; Lemke, M.; Quinlan, D.: AMR++, a parallel adaptive mesh refinement object class library for fluid flow problems; *Proceedings of the Symposium on Adaptive, Multilevel and Hierarchical Strategies*, ASME Winter Annual Meeting, Anaheim, CA, 1992.

[2] Bell, J; Berger, M.; Saltzman, J.; Welcome, M.: Three dimensional adaptive mesh refinement for hyperbolic conservation laws; *Internal Report*, Los Alamos National Laboratory.

[3] Brandt, A.: Multi-level adaptive solutions to boundary value problems; *Math. Comp.*, 31, 1977, pp. 333-390.

[4] Chandy, K.M.; Kesselman, C.: CC++: A Declarative Concurrent Object Oriented Programming Notation; California Institute of Technology, *Report*, Pasadena, 1992.

[5] Dörfer, J.: Mehrgitterverfahren bei singulären Störungen; *Dissertation*, Heinrich-Heine Universität Düsseldorf, 1990.

[6] Hempel, R.; Lemke, M.: Parallel black box multigrid; *Proceedings of the Fourth Copper Mountain Conference on Multigrid Methods*, 1989, SIAM, Philadelphia.

[7] Lemke, M.: Multilevel Verfahren mit selbst-adaptiven Gitterverfeinerungen für Parallelrechner mit verteiltem Speicher; *Dissertation*, Universität Düsseldorf, to appear in 1993.

[8] Lemke, M.; Quinlan, D.: Fast adaptive composite grid methods on distributed parallel architectures; *Communications in Applied Numerical Methods*, Vol. 8, No. 9, Wiley, 1992.

[9] Lemke, M.; Quinlan, D.: P++, a C++ Virtual Shared Grids Based Programming Environment for Architecture-Independent Development of Structured Grid Applications; *Lecture Notes in Computer Science*, No. 634, Springer Verlag, September 1992.

[10] Lemke, M.; Quinlan, D.: An Object-Oriented Approach for Parallel Self-Adaptive Mesh Refinement on Block Structured Grids; Proceedings of the 9th GAMM-Seminar on Adaptive Methods, Kiel, Germany, 1993; *Notes of Numerical Fluid Mechanics*, Vieweg, to appear.

[11] Lonsdale, G; Schüller, A.: Multigrid efficiency for complex flow simulations on distributed memory machines; *Parallel Computing*, 19, 1993, pp23 - 32.

[12] McCormick, S.: Multilevel Adaptive Methods for Partial Differential Equations; *Frontiers in Applied Mathematics*, SIAM, Vol. 6, Philadelphia, 1989.

[13] McCormick, S.; Quinlan, D.: Asynchronous multilevel adaptive methods for solving partial differential equations on multiprocessors: Performance results; *Parallel Computing*, 12, 1989.

[14] McCormick, S.; Quinlan, D.: Idealized analysis of asynchronous multilevel methods; *Proceedings of the Symposium on Adaptive, Multilevel and Hierarchical Strategies*, ASME Winter Annual Meeting, Anaheim, CA, Nov. 8 - 13, 1992.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	November 1993	Conference Publication

4. TITLE AND SUBTITLE
Sixth Copper Mountain Conference on Multigrid Methods

5. FUNDING NUMBERS
WU 505-59-53-01

6. AUTHOR(S)
N. Duane Melson, T. A. Manteuffel, and S. F. McCormick, editors

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
NASA Langley Research Center
Hampton, VA 23681-0001

8. PERFORMING ORGANIZATION REPORT NUMBER
L-17275

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
National Aeronautics and Space Administration, Washington, DC 20546-0001; Air Force Office of Scientific Research, Bolling AFB, Washington, DC 20338; the Department of Energy, Washington, DC 20585; and the National Science Foundation, Washington, DC 20550.

10. SPONSORING/MONITORING AGENCY REPORT NUMBER
NASA CP-3224
Part 1

11. SUPPLEMENTARY NOTES
Organizing Institutions: University of Colorado at Denver, Front Range Scientific Computations, Inc., and the Society for Industrial and Applied Mathematics.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Unclassified–Unlimited

Subject Category 64

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The Sixth Copper Mountain Conference on Multigrid Methods was held on April 4–9, 1993, at Copper Mountain, Colorado. This book is a collection of many of the papers presented at the conference and so represents the conference proceedings. NASA Langley graciously provided printing of this document so that all of the papers could be presented in a single forum. Each paper was reviewed by a member of the conference organizing committee under the coordination of the editors.

The multigrid discipline continues to expand and mature, as is evident from these proceedings. The vibrancy in this field is amply expressed in these important papers, and the collection clearly shows its rapid trend to further diversity and depth.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Multigrid; Algorithms; CFD	368
	16. PRICE CODE A16

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	